03

A busca binária

Transcrição

Vamos ver passo a passo como o nosso algoritmo se comportou? Veremos como ele dividiu o *array*...

Daremos um `System.out` e, no momento em que formos buscar, ele falará: "Buscando" + buscando + " entre " + de + " e " + ate);

```
private static int busca(Nota[] notas, int de, int ate, double buscando) {
    System.out.println("Buscando" + buscando + "entre" + de + " e " + ate);
    int meio = (de + ate) / 2;
    Nota nota = notas[meio];
    if(buscando == nota.getValor()) {
        return meio;

    }
    if(buscando < nota.getValor()) {
        return busca(notas, de, meio -1, buscando)
    }
        return busca(notas, meio + 1, ate, buscando);
}
```

Vamos rodar o algoritmo e o resultado será?

Buscando 9.3 entre 0 e 9

Buscando 9.3 entre 5 e 9

Encontrei a nota em 7.

jonas 3.0

andre 4.0

mariana 5.0

juliana 6.7

guilherme 7.0

carlos 8.5

paulo 9.0

lucia 9.3

ana 10.0

Ele buscou o 9.3 entre 0 e 9. Logo, ele percebeu que o elemento estava para direita, por isso, ele buscou entre 5 e 9. Depois, ele encontrou a nota na posição 7. Foi um processo muito rápido. Ele fez duas comparações, na segunda, ele encontrou o elemento.

Agora tentaremos encontrar outro valor: **6.7**.

```
ordena(notas, 0, notas.length);
int encontrei = busca(notas, 0, notas.length, 6.7);
```

Ao rodarmos o algoritmo novamente, esse será o resultado:

```
Buscando 6.7 entre 0 e 9

Buscando 6.7 entre 0 e 3

Buscando 6.7 entre 2 e 3

Buscando 6.7 entre 3 e 3

Encontrei a nota em 3

jonas 3.0

andre 4.0

mariana 5.0

juliana 6.7

guilherme 7.0

carlos 8.5

paulo 9.0

lucia 9.3
```

Neste caso ele buscou a nota 6.7 entre 0 e 9. Ele percebeu que o elemento era menor, então eliminou a parte da direita e buscou apenas na primeira metade menos o elemento do meio, de 0 até 3. Da parte restante, ele observou que era maior do que o meio. Então, ele ignorou o elemento 1. Depois, ele viu que estava mais para a direita que o elemento 2 e buscou entre o 3 e o 3. Então ele encontrou! A nota estava na posição 3.

Quando não encontra um elemento

É o momento de fazermos mais alguns testes com o nosso código. Nós já testamos e imprimimos o que ele estava buscando, com dois valores que existiam no *array*. Agora vamos testar dois valores que não existem.

Primeiro, testaremos com o valor **3.7**.

```
ordena(notas, 0, notas.length);
int encontrei = busca(notas, 0, notas.length, 3.7);
```

Ao rodarmos o meu programa ele irá apresentar uma mensagem de erro. O programa chamou o `busca`, depois o chamou novamente, e repetiu infinitamente até que explodiu. Por que ele chamou o `busca` infinitas vezes? Veremos o que aconteceu.

Ele buscou entre 0 e 9, mas não encontrou e dividiu pela metade. Depois, buscou entre 0 e 3, mas não encontrou novamente. Ele dividiu pela metade e buscou entre 0 e 0. Também não encontrou. Então ele ficou confuso, porque teve que dividir 0 na metade. Não fazia mais sentido continuar... Quando temos apenas um elemento, só existem duas opções: ele é o que buscamos ou não é. Não adianta continuar a dividir por dois. Então, se o número que temos entre `de` e `ate` for igual ou menor do que 1, não adianta continuar com a busca. No entanto, é o que continuamos fazendo. Quando tínhamos um elemento, seguíamos com a procura. Mas quando chegamos a esse ponto, temos apenas um - assim, ele é ou não é o que estamos buscamos. Se chegamos a zero elementos, ele não é. Então, nós não nos preocupamos com o caso de que o elemento não seja encontrado no nosso *array* (o caso do `return -1`).

Como você alteraria essa parte do algoritmo, para verificar se a nota não foi encontrada?

```
private static int busca(Nota[] notas, int de, int ate, double buscando) {
    System.out.println("Buscando" + buscando + "entre" + de + " e " + ate);
    int meio = (de + ate) / 2;
    Nota nota = notas[meio];
    if(buscando == nota.getValor()) {
        return meio;

    }
    if(buscando < nota.getValor()) {
        return busca(notas, de, meio -1, buscando)
    }
        return busca(notas, meio + 1, ate, buscando);
}
```

O *array* ficou muito pequeno, mas ele não encontrou o elemento.

Corrigindo a busca por divisão

Como podemos corrigir o nosso algoritmo para que, tanto no caso de sobrar um elemento, ou se cairmos no caso de que o `de` seja maior do que o `ate` ele funcione. No exemplo em que testamos a nota 3.7, o resultado foi:

```
Buscando 3.7 entre 0 e 9

Buscando 3.7 entre 0 e 3

Buscando 3.7 entre 0 e 0

Buscando 3.7 entre 1 e 0

Buscando 3.7 entre 1 e 0

Buscando 3.7 entre 1 e 0
```

Buscando 3.7 entre 1 e 0

Ele buscou de 1 até 0, não encontrou a nota e ficou louco.

Então, se (`if`) o `de` for maior que `ate` , significa que não encontramos nada e o código irá retornar -1.

```
private static int busca(Nota[] notas, int de, int ate, double buscando) {
    System.out.println("Buscando " + buscando + " entre " + de + " e " + ate);
    if(de > ate) {
    Nota nota = nota[meio];
    if(buscando == nota.getValor()) {
        return -1;
    }
   }
}
```

Iremos salvar o arquivo e depois, rodamos o programa. O resultado será:

Buscando 3.7 entre 0 e 9

Buscando 3.7 entre 0 e 3

Buscando 3.7 entre 0 e 0

Buscando 3.7 entre 1 e 0

Encontrei a nota em -1.

jonas 3.0

andre 4.0

mariana 5.0

juliana 6.7

guilherme 7.0

carlos 8.5

paulo 9.0

lucia 9.3

ana 10.0

Ele buscou 3.7 entre 0 e 9, em seguida entre 0 e 3, e depois, entre 0 e 0. Quando ele buscou entre 1 e 0, ele ficou confuso e concluiu que não encontrou. Por isso, ele devolveu que encontrou a nota na posição -1. Ou seja, precisamos adicionar um `if` ao nosso código: se (`if`) `encontrei` menor ou igual a 0, então descobri a nota.

```
ordena(notas, 0, notas.length);
int encontrei = busca(notas, 0, notas.length, 3.7);

if(encontrei >= 0) {
    System.out.println("Encontrei a nota em " + encontrei + " . ");
}
```

Senão (`else`), um `System.out` irá dizer: "Não encontreio a nota");

```
if(encontrei >= 0) {
    System.out.println("Encontrei a nota em " + encontrei + " . ");
} else {
    System.out.println("Encontrei a nota");
}
```

Ao rodarmos novamente, o programa irá imprimir:

Buscando 3.7 entre 1 e 0

Não encontrei a nota

jonas 3.0

andre 4.0

mariana 5.0

juliana 6.7

guilherme 7.0

carlos 8.5

paulo 9.0

lucia 9.3

ana 10.0

Ele informa que não encontrou a nota 3.7. Se testarmos com uma nota que existe no *array*, como por exemplo a nota 9, o programa irá imprimir o seguinte resultado:

Buscando 3.7 entre 1 e 0

Encontrei a nota em 6.

jonas 3.0

andre 4.0

mariana 5.0

juliana 6.7

guilherme 7.0

carlos 8.5

paulo 9.0

lucia 9.3

ana 10.0

Ele dirá que encontrou a nota na posição 6.

A Busca binária

Nós implementamos uma busca mais inteligente do que a busca tradicional - que varre todo o nosso *array*. Isto significa que quando tínhamos 100 elementos, a busca tradicional passava com um `for` pelos 1000 elementos. Se tivéssemos 1000 elementos, o `for` passaria pelos 1000 elementos. Se tivéssemos *n* elementos, o `for` passava por *n* elementos. Analisando o algoritmo da busca tradicional, ele era *O(n)* e passava por todos os elementos. Era uma busca do tipo **linear**.

Este algoritmo será rápido, enquanto trabalharmos com 100 ou 1000 elementos. Mas com 1 milhão, ele começará a ser lento, e por isso, buscamos outra maneira de fazer a busca. Nós procuramos uma forma em que o *array* já fosse ordenado.

Quando temos uma lista com os elementos ordenados, podemos direcionar nosso olhar para um parte dela. Se temos um *array* organizado, podemos quebrá-lo em duas partes, observar cada uma delas e depois, realizar a busca em apenas uma.

Essa é a **busca binária**: ela irá buscar em um dos dois pedaços do *array*.

Falta analisar a rapidez da busca binária... Temos a sensação de que ela é mais rápida. Porém, quão mais rápida ela é?